Adapting Transformers to Text Classification

Alexandre Matton
Stanford University
alexmt@stanford.edu

Abstract

The Transformer (Vaswani et al., 2017) is known for having produced state-of-the-art results in several NLP tasks when it was released in 2017. It heavily relies on attention, and gets rid of all the recurrent structures that were found in RNN architectures, which were until now ubiquitous in the fields. However, it was mainly tested on sequence-to-sequence tasks, notably translation. In this paper, we show that the Transformer can also be adapted to text classification and give scores that are not too far from what can be currently found in the literature. We also derive several architectures by replacing components. This allows us to provide interesting insights about how this architecture works in such tasks. Finally, we test it at the character-level and compare it with what we find at the word-level. We show that scores are generally lower at the character-level, but they can be improved with some slight changes, notably convolutions. These modifications also make training faster and more stable.

1 Introduction

After the first version of the Transformer was released in (Vaswani et al., 2017), a lot of researchers started working on it and finding ways to adapt it to different tasks, or datasets with different sizes.

The initial Transformer is a neural network architecture with an encoder and a decoder. The encoder's main part is a self-attention block, in which each input token can attend to all the other input tokens. The input tokens are modified through this attention layers. On top of it were placed some layer normalization (Lei Ba et al., 2016) blocks, residual connections like in (He et al., 2016) and feed-forward layers. The layer normalization makes and residual connections are here to make the training more stable and more efficient. The feed-forward layers adds more non-linearity and complexity to the overall structure.

This whole sets of sub-blocks can then be repeated several number of times. The original paper stacks up to 6 layers, but recently some much deeper networks have appeared. For instance, (Al-Rfou et al., 2018) use an architecture with 64 layers in total. The decoder architecture is approximately similar. The self-attention layer is replaced by a masked self-attention on the output, which prevents words from attending to words that appear next in the sentence, and an output-to-input classic attention layer. The Transformer uses other tricks, such as multi-headed attention, even if the usefulness of this part is actually questionable (Paul Michel, 2016).

This architecture can be modified to be more task-specific. For instance (Liu et al., 2018) introduce some changes to make it work on very long articles (with inputs containing several thousands of tokens). One of the tricks they implement consists in projecting the keys and values of the attention into a space with a low dimension.

Another interesting architecture is the one introduced in (Wu et al., 2019). The authors realized that for text summarization, the most important features to take into account were the short-term dependencies. The original Transformer has trouble promoting them over long-term dependencies as the notion of distance between words is represented less clearly, as it is only done through positional encoding. Hence, they figured out that local architectures were better, and implemented an architecture which is intermediairy between usual convolutions and self-attention. This led to much better results in text summarization on the datasets they tested. Based on this idea, we also implemented a local transformer, which cuts the articles into little chunks of fixed size and runs self-attention on each chunk. This led to an improve-

ment in both the results and the speed of the overall algorithm for long articles.

To our knowledge, less work of this kind was done for text classification. We show in this paper that the Transformer architecture is still relevant for such task. Moreover, we take inspiration from what has been done previously to improve the architecture. First of all, the decoder part is not mandatory anymore. We replace it by other simpler structures, such as a max pooling layer, or by introducing a new token at the beginning of each sentence. We then use the final embedding of this new token as input to the softmax classifier, as it is done in (Devlin et al., 2018). We show that these tricks have a positive impact over the speed of the Transformer, while limiting the accuracy loss. Finally we adapt the Transformer to character-level. With big enough datasets, we show that it scores close to the word-level. However, this requires the use of initial convolutions, and some additional changes, which take care of the feature extraction phase.

2 Related work

Character-level modeling of natural language text is challenging, for several reasons. First, the model must learn a large vocabulary of words from scratch. Second, natural text exhibits dependencies over long distances of hundreds or thousands of time steps. Third, character sequences are longer than word sequences and thus require significantly more steps of computation.

There has been several succesful attempts to work at this level with convolutions. One pioneering paper is (Zhang et al., 2015). They showed that they could get better results than the traditional word-level methods with big datasets. The convolutional networks they use were 9 layers deep, with 6 convolutional layers and 3 fully-connected layers. Convolutional kernels of size 3 and 7 were used, as well as simple max-pooling layers. Another interesting aspect of this paper is the introduction of several large-scale data sets for text classification. It shows that character-level needs more data than word-level to work, so it is mostly effective when the size of the dataset exceeds half a million train samples.

Another important paper which extends the previous one is (Conneau et al., 2016). They start from the same idea but manage to train a neural network which is much deeper. They mainly took inspiration from computer vision where significant improvements have been reported with these kinds of networks. They eventually show that performance improves with increased depth, using up to 29 convolutional layers. To make the training stable and efficient, they use tricks that are ubiquitous in computer vision: batch normalization (Ioffe and Szegedy, 2015) and Resnets (He et al., 2016). Moreover, they progressively increase the size of the embeddings while decreasing the total number of tokens via local pooling. One difference with the previous paper is that they only use small convolutions with size 3. As their architecture is really deep, this isn't a major problem, because information can still flow from one side of the input sentence to the other.

Up to now, little work combining Transformer and character-level has been done. One recent paper which promotes such an architecture is (Al-Rfou et al., 2018). They mainly work on the language modeling task so they do not need a decoder. Instead, they use the last token embedding to predict the next character. Hence, the self-attention they use is masked in order to prevent words to attend for the next ones. By adapting a little bit the transformer architecture and adding a lot of auxiliary losses between each layer, they manage to get a very stable learning. It makes it possible for them to stack up to 64 layers. They report state-of-the-art results in bits per character (bpc) on several datasets, even if the perplexity is still much bigger than with word-level models. However, they show that the system they built is quite robust. It is able to copy over long distance (up to 400 characters aparts), and promotes actual English words over non-existent words.

Another concept which is gaining interest among the NLP community is the level of subword units. It first began with the introduction of BPEs (Byte-pair encodings) (Sennrich et al., 2015). This algorithm comes from a data compression technique that iteratively replaces the most frequent pair of bytes in a sequence with a single, unused byte. It can be adapted for word segmentation. Instead of merging frequent pairs of bytes, the idea is to merge characters or character sequences. It helps to process rare words as it divides them into smaller units. These small units appear much more frequently than the words they come from, so their embeddings are more coherent as more training data is available. (Devlin et al., 2018) also

use a similar kind of subword representation for rare words. They push this direction even further as they base their training on such representations. This has been shown to lead to great results, as it brings an efficient solution to the recurring problem of rare words.

Finally, a last interesting direction to explore is to work on several level of representations at the same time. This has been done notably in (Peters et al., 2018) where each word representation is based on several convolutional neural networks with filters of different sizes. This method was also highly succesful and led to some breakthroughs in several NLP tasks when it was released.

The great point that is shared between all these contextual word representation systems is that they can be used in any system for almost any task. Moreover, they almost constantly increase the efficiency of the system in which they are integrated. Finally, these systems are perfect candidates for transfer learning, which makes them efficient even for small datasets.

3 Data

We decided to work on two classification datasets.

The first dataset is Stanford Sentiment Treebank (Socher et al., 2013). It is a very small dataset with short sentences. It contains 11855 sentences, with 8544 sentences for the train set, 1101 for the dev and 2210 for the test set. Moreover, after tokenization, the average number of tokens per sentences in 19 and the average number of characters per sentence is 104. Working on a dataset with short sentences makes sense, as there is always a need to better capture sentiment from short comments, such as Twitter data, which provide less overall signal per document. Each sentence is labeled with a number between 1 and 5 which corresponds to the sentiment of the sentence. The greater the label, the more positive the sentence. We could have chosen the binary classification problem, which is more usual, but as the results are very high in this case, it makes less room to see improvements.

The second dataset is AG-News (Zhang et al., 2015). The initial size of the train set in 120,000 and the size of the test set is 7600. We decided to take 6000 randomly chosen sentences from the train set to create a validation set. Moreover, the average post-processing size of each input is 41 tokens, or 245 characters. Hence, this dataset is much bigger than SST and contains longer sentences. It is actually closer to the usual size of datasets that people use for this type of task. The dataset contains 4 classes, which relate to the category of the articles (sport, world news, business, science/tech).

Both datasets are very popular. A lot of researchers have already worked with them by running their models on them so there exists well-defined baselines. They all deal with accuracy. For SST, the recent results are around 50% accuracy. The SOTA result comes from (Patro et al., 2018). They claim 64.4% accuracy on this task, which is more than 8 points higher than the second result. For AG-news, the SOTA results are 95.05% and come from (Sachan et al., 2018) which uses a classic architecture LSTM but introduces new losses to train it and get impressive results.

For each dataset, we applied the same preprocessing procedure. We lowercased everything to limit the size of the vocabulary. Moreover, we replaced all figures by a new token NUMTOKEN as we suspected that what the figure values do not bring useful information for text classification. Moreover, it would be really hard to create a tailored embedding for each figure that may appear in the text. For the AG-news dataset, I removes the name of the newspapers in the title and articles that contained it as it would give too much information about the category of the article.

As we will see, the difference of size between the datasets makes them react very differently to neural networks and hyperparameters. For instance, some models might work well on one dataset and not produce anything on the other one. Hence, we deemed them complementary, and for the purpose of the papers we decided that it would be interesting to compare the models that we run on them.

4 Models

4.1 Replacing the decoder

The first objective of the project was to see if it was possible to make use of the fact that sentence classification is not a sequence-to-sequence task so there is no need for a decoder.

We built 2 different structures to adapt the Transformer to text classification, without modifying the encoder, and compared them.

As a baseline, we used the initial architecture of the Transformer. To make it work with text categorization, we can train it so that it outputs only one token (+ the end-of-sentence token), which corresponds to the category. To do that, for each input sentence in the train set, we use the category number as the output sentence. (Hence the size of the output vocabulary is exactly the number of categories + the end-of-sentence token). The Transformer rapidly understands that it should output only one word everytime.

For the two structures that we built, we completely got rid of the decoder and replaced it by some new layers.

For the first new model, we added a max pooling layer on top of the encoder. For each dimension, it retrieves the greatest value among all the output tokens of the encoder. Hence, the max pooling layer outputs a new token of the same dimension as the input tokens. We then feed it into the softmax classifier to get the category with the highest probability.

The second model is actually inspired by the BERT paper (Devlin et al., 2018). Before running each input sequence to the encoder, we add a first token with a fixed embedding in front of it. Then, we run the encoder on this new sequence. In the end, the first token has a completely new embedding, that we directly use as input to the final softmax classifier.

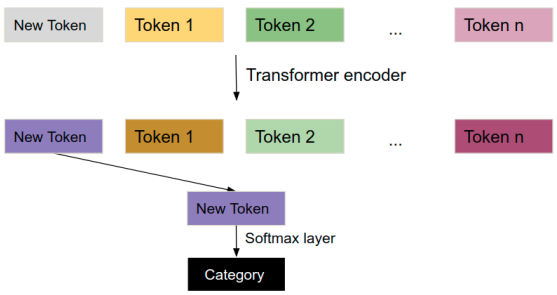

Figure 1: New architecture based on adding a new input token

4.2 Character-level Transformer

4.2.1 Convolutions

We rapidly understood that the vanilla Transformer has trouble working at the character-level. This may be due to several reasons. Our main explanation is that in the Transformer, except for the position encodings, nothing really makes the embeddings of the same character very different, when it appears several times in the same sentence. Hence, it is hard for the Transformer to understand that these occurrences don't appear in the same context. As characters do not have any meaning without their context, contrary to words, the vanilla Transformer can not work well in this setup.

We found two ways to fix this issue. The first way is to use convolutions on the initial character embeddings before feeding them to the classifier. Indeed, it has been shown in several papers, notably in (Zhang et al., 2015) that convolutions are well adapted for this level of precision. What they bring here to the original architecture is a first feature extraction phase. Hence, the token embeddings which are coming from the convolutions and going through the transformer are more context-dependent. Each convolution was done with a small kernel (3) and we added padding such that the number of input and output tokens stay the same. To add non-linearity to this phase, we also put some ReELU layers between each convolution. We show in section 5 that this new architecture works much better and we obtain much more consistent results, with a more stable training phase.

We also tried using a more complex architecture presented in (Conneau et al., 2016) by interleaving Batch normalization layers (Ioffe and Szegedy, 2015) between each convolution. This is a widely used trick in computer vision, which is supposed to make the training more stable. We found that it mainly lowered our results. The main reason may probably be that the biggest batchsize that we could use to make the whole architecture fit in the memory was too small to make this technique really effective.

The complexity of convolutions is $\mathcal{O}(n \times d^2)$. This can in theory be bigger than the one of self-attention, so convolutions may slow down the whole architecture. However, as there exists only very few characters, we don't need to embed them into a big dimensional space. Some speed tests have been run and they showed than the convolutions were not taking a big amount of the computation time (between 4% and 8%).

4.2.2 Local Transformer

The second method which we can use to make the vanilla Transformer work at the character-level corresponds to introducing a small change in the architecture. This architecture is what we call the Local Transformer.

The local transformer divides the input sequence into chunks of fixed-size which are processed independently by the encoder. Hence it is a more context-aware structure than the original one, as it forces the characters only to attend to close ones through this new "local attention".

Moreover, with a fixed-size window attention, the cost becomes linear while retaining many advantages of self attention. Indeed, with attention windows of size k, which is supposed to be much smaller than n, the local transformer replaces one self-attention on the whole input by n/k self-attentions on k tokens. The global complexity of the new block is $\mathcal{O}(k^2 \times d \times n/k) = \mathcal{O}(n \times k \times d)$, instead of $\mathcal{O}(n^2 \times d)$. The architecture is represented in Figure 2.

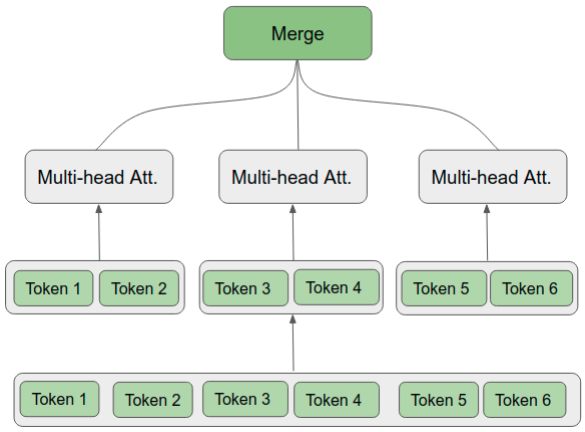

Figure 2: Local Attention

Hence, this simple trick helps solve two problems. Moreover, it can easily be combined with the convolutions presented just above as these two modifications don't act on the same part of the encoder. This is actually what gave us the best results/speed.

5 Results

5.1 Model parameters

The Transformer is an architecture which is hard to make work without the right hyperparameters and learning rate schedule. Because we didn't have a lot of credits/time to run our experiments, we couldn't do a large hyperparameter search so we started from the parameters of the original papers. The change we made concerning the values of the hyperparameters were mostly due to our modeling intuition.

Hence, for word-level, we used a Transformer with 6 blocks for both encoder/decoder, 8 attention heads, a word embedding dimension of 512, and 2048 in the feed-forward network on top of the encoder. We took a batch-size of 4096 words (the biggest that we could do). To make the training more stable and efficient we applied classic methods/tricks: Adam optimizer (Kingma and Ba, 2014), 8000 warmup-steps, Noam decay (which corresponds as it is said in (Vaswani et al., 2017) to increasing the learning rate linearly for the first warmup_steps training steps, and decreasing it thereafter proportionally to the inverse square root of the step number).

For character-level we keep the same parameters, except that we divide all embedding dimensions by 4 and multiply the batch-size by 2. We also found it useful to divide the overall learning-rate by 10.

5.1.1 Dropout

We decided to use dropout as our main regularization method, as it is done in the original paper.

We kept a dropout of at least 0.1 in each one of our model. When training at the word-level, we noticed that the training was extremely fast, and convergence was sometimes reached before 1 epoch. To make up for this fact and make full use of the available data, we tried giving to the model an aggressive dropout rate of 0.7. It worked and gave the model slightly better results at the expense of a much longer training phase.

For character-level, the training is already very long, so we didn't feel the need to modify the dropout rate.

5.1.2 Initial embeddings

For word-level Transformer, we also tried adding initial word embeddings from GloVe (Pennington et al., 2014) to compare the models with and without. We found interesting results that we described in part 6.

5.1.3 Parameters for convolutions/local transformer

For the local transformer we took a kernel size of 50 on AG-news, which approximately corre-

sponds to 10 words. We considered that this could provide enough context for characters to train, while being short enough to make the architecture work significantly faster. Texts are shorter on SST, so changed k from 50 to 30 in this dataset.

In the models with initial convolutions, we added 6 convolutions of kernel size 3, We didn't have the time to test other numbers of convolutions, but considered 6 to be a reasonable value. This could be an interesting parameter to explore later.

5.1.4 Results

The most important results that we got are listed in the tables below. For each case, we gave the number of steps T necessary to get the best validation score and the corresponding score on the test set. N.R. is for "not relevant", N.C. for "not computed".

For the first table, all the results are computed at the word-level with GloVe embeddings.

Architecture	Results
AG-news	
Vanilla Transformer	s = 91.6, T = 1400
Max-pooling	s = 91.8 , T = 2000
New token	s = 91.7, T = 2000
SST	
Vanilla Transformer	s = 42.7, T = 1400
Max-pooling	s = 32.7 , T = 3000
New token	s = 40.5, T = 500

Figure 3: Results of architectures without decoders

6 Analysis

6.1 Decoder replacement

All methods seem to approximately give the same results on AG-news, and for SST the Max-pooling method is behind the two others. The decoder seems to be slightly more efficient than the "new token" trick. However, as SST is a very small dataset, the differences between these two classifiers are not very relevant.

From these results, one can infer that in text classification, most information is actually contained in the encoder. The decoder is just a transition towards the final output, but it is not vital in itself. Hence, it seems to be a good idea to replace the decoder by the "new token" trick in this settings. Indeed, we found out that it increases the speed of each iteration of the Transformer by approximately 28 %. (We also found out that the speed improvement is exactly the same with max-pooling phase).

	Word-level	Character-level
SST		
Vanilla Transformer	s=34.5, T=1500	Couldn't train
+ GloVe	s=42.7, T=1000	N.R.
Convolutions	N.C.	s=35.3, T=4500
Dropout (0.7)	s=36.2, T=4000	N.C.
GloVe + Dropout (0.7)	s=37.5, T=1500	N.R.
Local + Convs	N.C.	s=38.2, T=31,000
AG-news (All results are computed with GloVe)		
Transformer	s=91.6, T=1400	s=78.1, T=150k
Local	N.C.	s=82.1, T=48k
Dropout (0.7)	s=91.8, T=2200	N.R.
Convolutions	s=91.4, T=1500	s=88.1, T=17k
Local + Convs	N.C.	s=88.2, T=12k

Figure 4: Results of word-level vs character-level

6.2 Word-level v.s. Character-level

For this comparison, we ran a lot of models with different sets of parameters. We got some insights from all the results we gathered:

First of all, the character-level transformer globally outputs results which are worse than the word-level one, even with the best architectures we found. However, we must admit that we didn't have the time/credits to run grid-search on hyperparameters. The set of hyperparameters that was used here was fine-tuned for the word-level Transformer for sequence-to-sequence tasks, so we can suppose that it is more adapted to words than characters. As we already know, result quality of deep networks can substantially vary according to the hyperparameters (and even experiments), so maybe with a better set of hyperparameters we could get much better results. A massive disadvantage of the character-level is that it cannot benefit from pre-trained embeddings such as GloVe, as the total number of characters is really small and their embeddings are too context dependent. Hence, they need to be trained from scratch for each dataset. This is essentially bad for very small datasets such as SST, over which the best models are the ones taking into account a

lot of external data.

For the word-level transformer, we observed that dropout helps to get better results. Text classification is a very easy task compared to other ones with a more complex output. Hence, the parameters tend to converge too fast, and the Transformer may not take advantage from the whole dataset. Big dropout rates fix that issue. As random neurons are removed from the structure, each neuron of the network has to be more aware of the whole structure. Moreover, it has been shown that when the number of parameters is big compared to the number of training samples, it is better to have a big dropout.
However, what worked best was starting with GloVe embeddings. This helps the network learn with some initial weights that already make sense, so it speeds up the training while giving better results. This is particularly noticeable for the SST dataset, which is very small. Indeed, in that case, adding external information to the system is particularly beneficial. For this dataset, the results we got are quite far from the State-of-the-Art, whereas they are much closer for AG-news. The main reason is that all the papers that manage to get a great score on SST extensively use transfer learning or pre-training. As our purpose was just to compare architectures that we built between themselves, we didn't want to spend too much time on pre-training.
As we can see in the results, using initial convolutions doesn't lead to better results at the word-level. Our explanation for that is that word embeddings already bring enough information so that the Transformer can process to feature extraction on its own. Moreover, context is less important than for characters.

For the character-level transformer, the first noticeable result is that the vanilla transformer doesn't even train on SST, and outputs pretty poor results on AG-news. The category it outputs more than 90% on the time on SST is the category number 4, which is the dominant one. Hence, it almost doesn't take into account the actual sentence. As it was explained in 4.2.1, our main hypothesis is that it doesn't manage to do extract features from the input, because characters don't make sense without their context and the Transformer works globally and not locally.

However, with convolutions, the results get much closer to what we can get at the word-level. Moreover, when replacing the encoder by its local equivalent, we see a slight improvement, and a faster convergence on AG-news. We got our best results when combining the two tricks. Even if the score is quite close to the one that we could get with convolutions only, the convergence is 42 % faster on the dataset in terms of number of steps, and more importanly we observed the Transformer processes the same amount of data almost 40 % faster too. These results give hope as we did not finetune neither the kernel size of the local transformer nor the number of convolution layers. An other important fact is that this optimized character transformer processes data up to 83 % faster than the word-level vanilla transformer. For training this is shadowed by the fact that character-level needs many more epochs to converge than word-level. However, at test time (which is the important measure for industry purposes), this gain is far from negligible.

For SST, the Local Transformer really helps to improve the score compared to a model with convolutions only. The convergence is slower in this case, but it is mainly because the overall architecture manages to generalize more than the one with only convolutions. Indeed, in the latter case, the validation score starts droping right after 4500 steps, which is small for character-level. The scores on train set and validation set are less correlated.

6.3 Training phases

The most general law is that the word-level transformer trains much faster than the character-level one. However, the necessary number of steps for the character-level architecture to converge is greatly reduced with convolutions. These two laws are verified by the learning curves that were obtained via Tensorboard. The number of steps to reach convergence is very different.

The y-axis of the figures below correspond to

$$y = \frac{accuracy + 1}{2}.$$

It is due to the fact that the transformer always outputs two tokens, the first one being the category and the second being the end-of-sentence token to

say that the prediction stops there. Hence, the accuracy is computed over these two tokens instead of just the first one.

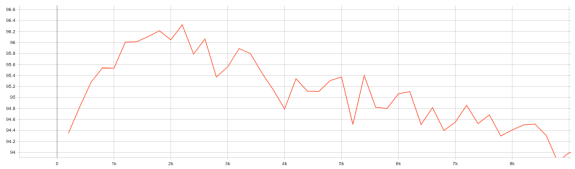

Figure 5: Learning curve for the normal transformer with GloVe embeddings trained on AG-news

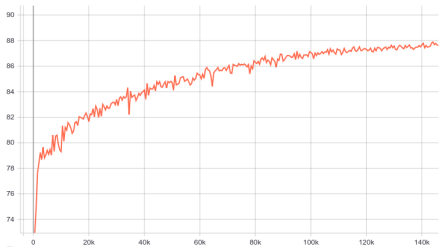

Figure 6: Learning curve for the char-level transformer trained on AG-news

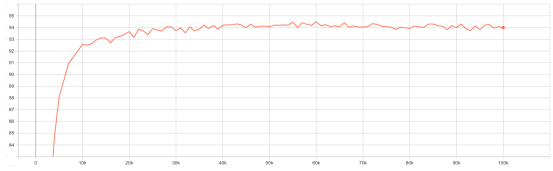

Figure 7: Learning curve for the char-level transformer trained on AG-news, with initial convolutions

The convergence of the char-level transformer is extremely slow as more than 100k steps are necessary to get a result which is not too far from the optimum. This corresponds to approximately 19h on a GPU K6000. The training on the normal transformer always take less than 1h.
However, the training phase seems much more stable at the character-level as we can see some clear overfitting in the word-level case.

7 Conclusion

As a conclusion, we showed that we could make the Transformer work for text-classification and get results that are not too far from the current literature. It is possible to work at the character-level for Transformer, but this requires a few extra tricks to make the training phase more consistent. With some initial convolutions and a more local architecture, we can produce results that are very close to the word-level Transformer. An interesting direction to explore would be to finetune the hyperparameters of this new system to make this architecture even more adapted to character-level. Eventually, one major advantage of character-level architecture is that they process data at a greater speed than word-level ones at test time, which is a non-negligible incentive for industry applications.

8 Code

The code we used to run all these experiments is available here: https://github.com/Nutemm/OpenNMT-py/blob/master/README.md

9 Acknowledgments

I would like to thank Professors Chris Potts for his precious feedbacks and advice throughout the project.

10 Authorship statement

I did this project alone, so all the code and results of the paper are coming from my own work. Even if I worked on a similar project during CS224N, I didn't take anything from it, and even recoded the architectures that we previously implemented on a new repository (OpenNMT (Klein et al., 2017)). I did that mainly because this repository seemed simpler and more appropriate for what I intended to do. Finally, the only common points between the two projects is that they are centered around the Transformer architecture, and they both make use of the local transformer architecture.

References

Rami Al-Rfou, Dokook Choe, Noah Constant, Mandy Guo, and Llion Jones. 2018. Character-level language modeling with deeper self-attention. *arXiv preprint arXiv:1808.04444*.

Alexis Conneau, Holger Schwenk, Loïc Barrault, and Yann Lecun. 2016. Very deep convolutional networks for text classification. *arXiv preprint arXiv:1606.01781*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on*

computer vision and pattern recognition, pages 770–778.

Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M. Rush. 2017. OpenNMT: Open-source toolkit for neural machine translation. In *Proc. ACL*.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450*.

Peter J Liu, Mohammad Saleh, Etienne Pot, Ben Goodrich, Ryan Sepassi, Lukasz Kaiser, and Noam Shazeer. 2018. Generating wikipedia by summarizing long sequences. *arXiv preprint arXiv:1801.10198*.

Badri N Patro, Vinod K Kurmi, Sandeep Kumar, and Vinay P Namboodiri. 2018. Learning semantic sentence embeddings using pair-wise discriminator. *arXiv preprint arXiv:1806.00807*.

Graham Neubig Paul Michel, Omer Levy. 2016. Are sixteen heads really better than one? *arXiv preprint arXiv:1905.10650*.

Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*.

Devendra Singh Sachan, Manzil Zaheer, and Ruslan Salakhutdinov. 2018. Revisiting lstm networks for semi-supervised text classification via mixed objective function.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008.

Felix Wu, Angela Fan, Alexei Baevski, Yann N Dauphin, and Michael Auli. 2019. Pay less attention with lightweight and dynamic convolutions. *arXiv preprint arXiv:1901.10430*.

Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. In *Advances in neural information processing systems*, pages 649–657.